Stack Caching
und
Register Allokation

Martin Stejskal

Student der technischen Universität Wien

e0225494@student.tuwien.ac.at

Kurzfassung

In diesem Dokument wird gezeigt, wie mit dem konkreten Problem der Register Allokation beim Stack Caching umgegangen werden kann. Es wird sowohl der bekannteste softwareseitige, wie auch der derzeit führende hardwareseitige Ansatz gezeigt. Die Thematik hat besondere Bedeutung da die vielverwendeten Interpreter am häufigsten, wegen der einfachen Implementierung, Stackmaschinen sind. Bei der zugrunde liegenden Hardware handelt es sich jedoch in den meisten Fällen um Registermaschinen. Einer der großen Ansätze zur Effizienzsteigerung von Interpretern ist somit das Stack Caching, als möglichst effektives Nutzen der vorhanden Register zur Verkürzung der Zugriffszeiten auf Argumente vom Stack, für Instruktionen zu implementieren.

Einführung

Im Bereich der Computersprachen haben sich zwei Richtungen des in eine maschinenverständliche Form Umwandelns von Quelltexten herauskristallisiert. Zum einen besteht der Ansatz des so genannten Kompilierens, wobei Programme welche im Quelltext vorliegen direkt oder in mehreren Schritten in vom Hauptprozessor interpretierbaren Binärcode übersetzt werden, und zum anderen gibt es die Richtung der Interpreter, bei denen Anweisungen von einem Programm, nämlich dem Interpreter, verstanden und so interaktiv zu maschinenverarbeitbaren Kommandos umgewandelt werden. Beide Wege haben sowohl Vor- als auch Nachteile. Die Stärken von Interpretern sind im Wesentlichen: Hohe Verbreitung und daher viele Erfahrungswerte, Einfachheit der Erstellung eines Interpreters, hohe Portabilität, hohe Flexibilität durch Programmerweiterungsmöglichkeit zur Laufzeit, dominant für vielseitige Programmiersprachen

Interpreter haben jedoch ein zentrales Problem, nämlich die Effizienz. Hier kann man drei Thematiken aggregieren: Instruktionsverarbeitung, Semantik der Programme und der Zugriff auf Argumente einer Instruktion

Instruktionsverarbeitung kann hierbei durch Threaded Coding [Bel73, Dew75] verbessert werden. Die Semantik von Programmen, kann durch Analyse und erstellen von Superinstruktionen zur Verbesserung beitragen. Dem dritten Problem, nämlich dem Zugriff auf Argumente einer Instruktion wird durch Cachen entgegengewirkt. Hier gibt es jedoch einen Spalt da meistens Register Maschinen als Hardware verwendet werden, Interpreter jedoch einfach als Stackmaschinen zu erstellen sind. Im Folgenden soll daher gezeigt werden wie ein möglichst effizient laufendes System geschaffen werden kann.

Register, der schnelle Speicher

Register sind kleine Speichereinheiten welche sich bei so genannten Registermaschinen gleich auf dem Chip des Mikroprozessors befinden. Diese können direkt von der CPU angesprochen werden und haben daher eine entsprechend kürzere Zugriffszeit als Cache oder gar der Hauptspeicher. Der Befehlssatz von Registermaschinen ist ebenfalls primär auf die Verwendung der internen Register ausgelegt, so sind logische sowie auch arithmetische Einheiten direkt an die Register angeschlossen. Ein weiterer Vorteil von Registern ist, dass aufgrund der meist wenigen Register, die Adressierung entsprechend wenig Aufwand bereitet und Code, sowie Instruktionen entsprechend kompakt sind. Da Register und Hauptspeicher jeweils eigenständig an der CPU hängen, kann zur selben Zeit auf den Hauptspeicher zugegriffen werden, während mit Registern gearbeitet wird. Weiters sind kleine Speichermengen wie Register leicht und billig zu vervielfachen und somit parallele umschaltbare Register Sets zu fertigen. Ein weiterer Punkt ist die Entlastung des Datenbusses, weil Register direkt an die CPU angebunden sind.

Den genannten Vorteilen stehen jedoch nicht günstige Eigenschaften von Registern gegenüber. Zunächst wäre zu erwähnen, dass Register bei Funktionsaufrufen komplett gesichert werden müssen, da es sonst zu Inkonsistenzen der Registerinhalte nach der Rückkehr vom Aufruf kommen kann. Dieser Effekt

macht sich besonders bei Programmiertechniken die mit vielen Funktionsaufrufen arbeiten bemerkbar. Da Register direkt von der CPU adressiert werden, können sie nicht wie jede andere Speicherstelle im Hauptspeicher angesprochen werden und erfordern daher eine separate Behandlung. Weiters kann ein Register meistens nur die Wortbreite der CPU aufnehmen, das heißt im Speziellen, dass in einem Register Hochsprachenkonstrukte wie Strukturen oder Zeichenketten nicht abgelegt werden können. Aus Compilersicht ist auch zu erwähnen, dass die Allokation von Registern einiges an Komplexität mit sich bringt. Compiler welche Register Allokation betreiben sollen, neigen eher zu Fehlern und schlechter Wart- sowie Erweiterbarkeit. Aus diesem Grund verzichten die meisten Compiler auf Register Allokation. Die Programmiersprache C [Ker78] bietet Entwicklern daher ein Schlüsselwort namens *register* an, welches die damit versehene Variable möglichst in einem Register der CPU ablegt.

Dennoch ist die Verwendung von Registern als Performanzoptimierung lohnend, wenn auf gewisse Bedingungen Acht gegeben wird. Mit bereits genannten Register Sets können Context Switches effizienter gestaltet werden. Einige Arbeiten haben sich bereits mit dem Thema der Effizienz von Registern auseinander gesetzt und festgestellt, dass 100 bis 1000 Register in Kombination mit dem oben genannten Register Sets nahezu alle performanztechnischen Hürden nehmen lassen. Zu bedenken bleibt jedoch noch, dass je größer die Anzahl der Register, desto länger und umfangreicher ist ein Context-Switch.

Cache statt Register

Unter dem Begriff Cache, oder auch Schattenspeicher genannt, versteht man sehr schnellen Speicher, welcher ebenfalls möglichst nahe dem CPU Kern platziert und meistens so wie auch Register direkt auf dem Silizium des Prozessorchips angeordnet wird.

Die wesentliche Vorteil gegenüber Registern ist, dass es sich dabei um Speicher handelt, so wie auch der Hauptspeicher. Damit sind die einzelnen Speicherstellen explizit adressierbar und auch mehrere hintereinander liegende Speicherstellen als gesamtes, wie es zum Beispiel für Arrays der Fall sein kann, verwendbar. Das ist jedoch der einzige wahre Vorteil von Cache gegenüber von Registern, da Cache im allgemeinen stets noch um einiges langsamer als Register ist. Weiters sind die Adressen der Cache-Speicherstellen meistens nur über Adressberechnungen erreichbar, und wenn es im Cache auch noch mehrere Eintrittsstellen von Cachelines gibt, dann ist noch ein Multiplexer der zwischen den Eintrittsadressen schaltet notwendig. Auch bei Verzicht auf mehrere Eintrittsstellen von Cachelines ist der Geschwindigkeitsunterschied signifikant. Weiters darf nicht übersehen werden, dass Caches bei Variablen welche gerade erst allokiert wurden, beim ersten Zugriff einen Miss liefern. Besonders bei Programmen mit hoher Dynamik was die Allokation von Speicher betrifft, haben Caches hier das Nachsehen. Register hingegen haben immer einen Hit!

Stack Caching in Registern

Hier sind zwei Ansätze zu nennen. Zum einen die so genannte Graphfär-

bung, zum anderen die Verwendung eines Ringpuffers.

Graphfärbung, ein softwaremäßiger Ansatz

Sowohl der Compiler als auch der Optimierer können eine unendliche Anzahl von Pseudoregistern für Variablen und temporäre Werte verwenden. Die Aufgabe der Register Allokation ist es in weiterer Folge diese Pseudoregister auf eine endliche Zahl tatsächlich vorhandener Register zu legen. Der heute bekannteste Ansatz zur Allokation von Registern ist das so genannte Graphfärben [CAC+81, Cha82, BCKT89], welches folgende Schritte beinhaltet.

Bilden des Interferenzen Graphs

Der Interferenzen Graph stellt die grundlegende Datenstruktur beim Graphfärben dar. Jedes Pseudoregister wird als ein Knoten dargestellt. Weiters befindet sich zwischen je zwei sich beeinflussenden Pseudoregistern eine Kante. Somit dürfen diese zwei benachbarten Knoten in weiterer Folge nicht die gleich Farbe bekommen.

Einfärben der Knoten

Sobald der Interferenzen Graph erstellt ist, können die Register allokiert werden. Dazu wird wie folgt vorgegangen: Ein ungefärbter Knoten wird gewählt und ihm ein Register, welches sich von den anderen eventuell bereits eingefärbten Knoten unterscheidet zugeteilt. Sollte dies nicht möglich sein, so muss spilling-code [Ert95], das heißt Code der einen Wert aus dem Register in den Hauptspeicher auslagert und dann einen Wert aus dem Hauptspeicher in das Register lädt, eingefügt werden.

An dieser Stelle sei erwähnt, dass folgende zwei Punkte bei diesem Algorithmus offen bleiben [AmErBeKr94]:
1.) In welcher Reihenfolge werden die Knoten eingefärbt?
2-) Welches Register wird einem Pseudoregister zugewiesen?

Die zwei Fragen scheinen auf den ersten Blick nicht von Interesse, jedoch darf nicht außer Acht gelassen werden, dass neben der Register Allokation auch ein Optimierung in Form von Umschlichten der Instruktionen stattfinden können soll.

Weitere Bemühungen zur Reduktion des Interferenzen Graphen bringen eine Kantenersparnis von 7%-24% und eine für das Scheduling von Instruktionen relevante Unabhängigkeit von 46%-100% [AmErBeKr94].

Das vorgestellte Verfahren des Graphfärbens liefert speziell in seiner verbesserten Form hochqualitativen Code, jedoch fällt bei einer Laufzeitanalyse das quadratische Verhalten auf. Dadurch ist dieses Verfahren für so genannte Just-in-Time-Compiler unbrauchbar [Wim04]. An gleicher Stelle wird das auf Linear-Scan getaufte Verfahren eingeführt, welches durch Optimierungen zu einer Verbesserung der Code-Qualität führt und somit die Ausführungsgeschwindigkeit um bis zu 30% steigert.

Ringpuffer Cache, ein hardwaremäßiger Ansatz

Bei diesem Ansatz wird die Problemstellung der Register Allokation nicht dem Kompiler zur Übersetzungszeit gelassen, sonder zur Laufzeit eine Zuteilung getroffen. Durch das Zusammenspiel von einer gut gewählten Art der Funktionsaufrufe, spezieller Hardware und ein eigenes Instruktionsset wird eine Art automatisches Binden von Speicher Adressen

und Maschinen Registern erreicht. Für den Kompiler erscheint das System als memory-to-memory Architektur, das heißt der Kompiler muss sich nicht um die Register kümmern. Folgende Schritte machen diesen Ansatz möglich.

Optimierte Funktionsaufrufe

Da Funktionsaufrufe besonders viel Einfluss auf die Performanz eines Programms haben, ist es besonders wichtig diese möglichst effizient zu gestalten. Ein Funktionsaufruf beinhaltet das Kopieren der Argumente, Speichern der Rücksprungadresse, Übergabe der Kontrolle an die aufgerufene Funktion, sowie das Allokieren von Speicherplatz für die lokalen Variablen der neuen Funktion. In Figure 1 ist ein heute typischer so genannter Stack Frame abgebildet. Der Stack vergrößert sich stets in Richtung der niedrigeren Adressen. SP der Stack Pointer zeigt stets auf eine freie Stelle im Speicher. Bei einem Funktionsaufruf (oder auch bei Auftreten eines Interrupt) wird an genau diese freie Stelle der aktuelle Programm Counter PC geschrieben. Diese Vorgehensweise hat den Vorteil, dass der Stack Pointer bei einem Funktionsaufruf nicht erst verändert werden muss. Der Programm Counter ist das einzige Maschinen Register, welches bei einem Aufruf gesichert werden muss.

Das Besondere an dieser Methode ist, dass der Stack Pointer nur bei Funktionsaufruf und Funktionsrücksprung verändert werden muss. Traditionelle Funktionsaufrufe halten lokale Variablen über und Argumente unterhalb des Stack Pointers, was dazu führt, dass der Stack Pointer zumindest für die Allokation der Argumente ein weiteres Mal verändert werden muss. Im Weiteren wurde bewusst auf die Funktionalitäten der Push und der Pop Funktionen verzichtet, da diese den Stack Pointer jedes Mal verändern und Instruktionen welche sich auf den SP beziehen somit bis zu dessen endgültiger Änderung warten müssen.

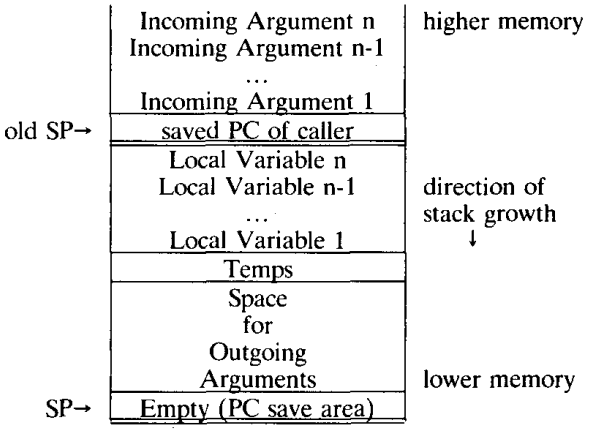

Figure 1. Stack Frame Layout

Ein Funktionsaufruf sieht nun wie folgt aus. Argumente des Funktionsaufrufs werden auf den Stack gelegt, Argument 1 auf die Adresse SP+1, Argument 2 auf SP+2 und so weiter. Hier ist anzumerken, dass nicht die traditionelle Push Funktion verwendet wird, damit der SP nicht jedes Mal beim Schreiben eines Arguments verändert wird. Die Call Instruktion des Funktionsaufrufs speichert die Rücksprungadresse auf den Stack. Weiters führt die erste Instruktion der neuen Funktion zum Allokieren des Stack Frames und zur Änderung des SP. Die return Instruktion der neuen Funktion führt ebenfalls zu einer Änderung des Stack Pointers auf die Stelle wo die Rücksprungadresse gesichert wurde. Dadurch wird der für die neue Funktion allokierte Stack Frame freigegeben und der Programm Counter kann auf den Wert des Speichers auf den der SP zeigt gesetzt werden um einen Rücksprung durchzuführen.

Spezielle Hardware

Um genügend Registerspeicher für mögliche Funktionsaufrufe zu haben, reichen die Maschinen Register üblicher

Prozessoren meist nicht aus. Daher verwendet dieser Ansatz ein eigenes Stack Caching Register Set, welches als Ring Puffer von zwei Registern, dem Maximum Stack Pointer MSP und dem Stack Pointer SP verwaltet wird. Der MSP hält dabei die höchste Adresse der Daten in den Stack Cache Registern und der SP die niedrigste. Die Stack Cache Register selbst sind gewöhnlicher, schneller Random-Access Speicher.

Zwei Schritte sind für das Allokieren von Register notwendig. Zuerst muss beim Holen einer Instruktion werden Adressberechnung der Form SP+Offset aufgelöst und in weiterer Folge immer die errechnete Adresse verwendet. Im zweiten Schritte können durch die geschickt gewählten Funktionsaufrufe als gewährleistet nehmen, dass sich der SP während der aufgerufenen Funktion nicht ändert. In weiterer Folge ist es daher möglich zu überprüfen, ob sich gewisse Daten im Stack Cache oder im Hauptspeicher befinden. Dies kann dadurch ermittelt werden, dass nachgesehen wird, ob sich eine gewisse Adresse zwischen dem SP und dem MSP befindet. Ist dies der Fall so befinden sich die Daten im Stack Cache und die Adressierungsart, die als Befehl in den Instruction Cache geschrieben wird, wird auf Registeradressierung gestellt.

Einer der Vorteile von virtuellen Adressen gegenüber der Verwendung von Offsets ist die Ersparnis einer Pipeline. Weiters wird bei jeder Referenz auf solch eine Funktion das Berechnen der Summe von Stack Pointer und Offset eingespart. Damit haben wir eine direkte Adressierung wie sie von Registern bekannt ist, jedoch verhält sich unser Cache nicht wie Register, welche nur als ganzes Wort gelesen und geschrieben werden können, sonder viel mehr wie gewöhnlicher Speicher. Das heißt, dass auch byteweise zugegriffen werden kann.

Integration im Instruction Set

Es werden vier Instruktionen für die Verwaltung des Stack Caches benötigt. Diese sind call, return, enter und catch. Die call Instruktion nimmt die Rücksprungadresse, legt sie auf dem Stack ab und springt zu der Zieladresse des Aufrufs. Das Ziel der call Instruktion muss eine enter Instruktion sein, welche Speicher für den neuen Stack Frame allokiert. Die return Instruktion wiederum gibt den Speicher vom Stack Cache frei und springt auf die Rücksprungadresse zurück. Die catch Instruktion ist stets das Ziel einer return Instruktion und sorgt dafür, dass nach dem Rücksprung wieder alle benötigten Daten im Cache vorhanden sind.

Die enter und die catch Instruktionen haben im Detail die Aufgaben, bei einem Funktionsaufruf den Stack Frame im Stack Cache unterzubringen und gegebenenfalls zu spillen um dies zu ermöglichen. Die catch Instruktion hat hier die Gegenaufgabe zur enter Instruktion. Sie sorgt dafür, dass nach dem Rücksprung alle lokalen Daten der Aufruferfunktion wieder im Stack Cache befindlich sind.

Die Verwendung eines Stack Cache ist am effizientesten, wenn der Stack dazu neigt, wenige Größenänderungen zu vollziehen. [Dit82]

Zusammenfassung

In diesem Paper wurde auf die Performanz-Problematik des Zugriffs auf Argumente von Interpretern eingegangen. Die zwei vorgestellten Ansätze, Graphfärbung und Ringpuffer Cache, zielen beide auf das Stack Caching ab. Zuerst wurden die Themen, *Was ist ein Register* und *Warum besser Register statt Cache verwenden*, erläutert. Im Weiteren wurden die beiden Ansätze erklärt. Bei der Graphfärbung übernimmt der Kompiler zur Übersetzungszeit die Analyse des Quelltextes und weist mittels des Graphfärbe-Algorithmus einzelnen Stackelementen Register zu. Da diese Variante in seiner Urform exponentielle Laufzeit aufweist, wurde noch eine verbesserte Variante welche lineares Laufzeitverhalten hat gezeigt. Der zweite Ansatz setzt spezielle Hardware, ein spezielles Instruktionsset sowie einen geschickten Funktionsaufruf voraus. Die Idee dahinter ist ein Ringbuffer, welcher das Stack Caching Register Set darstellt. Für den Kompiler ist dieser Ringbuffer transparent und die Register Allokation findet zur Laufzeit statt. Man erzielt damit eine Perfomanz wie mit Registern, ohne den Kompiler zu belasten.

Referenzen und verwandte Arbeiten

[AmErBeKr94] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, Andreas Krall. Dependence-Conscious Global Register Allocation. *Programming Languages and System Architectures*, Springer LNCS 782, 1994, pages 125-136

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370-372, 1973.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):45-57, 1981. Reprinted in [Sta90].

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82 [SIG82]*, pages 98-105.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275-284, 1989.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501-536, October 1990.

[Bri92b] Preston Briggs. Register Allocation via Graph Coloring. *PhD thesis*, Rice University, Houston, 1992

[Dew75] Robert B.K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330-331, June 1975.

[Ert95] M. Anton Erl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315-327, 1995.

[GoHs86] J.R. Goodman, W.C. Hsu, *On the use of registers vs. cache to minimize memory traffic*, Proceedings of the 13th annual international symposium on Computer architecture, p.375-383, June 02-05, 1986, Tokyo, Japan

[Ker78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

[SIG82] *SIGPLAN '82 Symposium on Compiler Construction*, 1982.

[Wim04] Christian Wimmer, Linear Scan Register Allocation for the Java HotSpotTM Client Compiler, *A thesis submitted in partial satisfaction of the requirements for the degree of Master of Science (Diplom-Ingenieur)*, Supervised by o.Univ.-Prof. Dipl.-Ing. Dr. Hanspeter Mössenböck, Institute for System Software Johannes Kepler University Linz, August 2004.

[Dit82] David R. Ditzel, Register allocation for free: The C machine stack cache, ACM SIGARCH Computer Architecture News, Volume 10, Issue 2, March 1982